The Effect of Flexible Parsing for Dynamic Dictionary Based Data Compression *

Yossi Matias[†] Nasir Rajpoot[‡] Süleyman Cenk Ṣahinalp[§]

Abstract

We report on the performance evaluation of greedy parsing with a single step lookahead, denoted as flexible parsing. We also introduce a new fingerprint based data structure which enables efficient, linear time implementation.

1 Introduction

The most common compression algorithms are based on maintaining a dynamic *dictionary* of strings that are called *phrases*, and replacing substrings of an input text with pointers to identical phrases in the dictionary. Dictionary based compression algorithms of particular interest are the LZ78 method [ZL78], its LZW variant [Wel84], and the LZ77 method [ZL77] which are all asymptotically optimal for a wide range of sources.

Given a dictionary construction scheme, there is more than one way to *parse* the input, i.e., choose which substrings in the input text will be replaced by respective *codewords*. Almost all dynamic dictionary based algorithms in the literature use

[*] Part of this work was presented in Workshop on Algorithmic Engineering, Saarbrucken, Germany, 1998.

[†] Computer Science Dept., Tel-Aviv University; `matias@math.tau.ac.il`. Research supported in part by an Alon Fellowship, by the Israel Science Foundation founded by The Academy of Sciences and Humanities, and by the Israeli Ministry of Science.

[‡] Mathematics Dept., Yale University; Computer Science Dept., Warwick University; `rajpoot@noodle.med.yale.edu`.

[§] Computer Science Dept., Warwick University; Center for BioInformatics, Univ. of Pennsylvania; `cenk@dcs.warwick.ac.uk`. Research supported by NATO research grant CRG-972175 and ESPRIT LTR Project no. 20244 – ALCOM IT.

greedy parsing which is fast and can be applied on-line. However, it usually results in far from optimal parsing/compression: for the LZW dictionary method, there are strings T which can be (optimally) parsed to some m phrases, for which the greedy parsing obtains $\Omega(m^{3/2})$ phrases ([MS99]; a similar result for static dictionaries is in [GSS85]).

In [Hor95] it was demonstrated that the compression achieved by LZW algorithm on some standard benchmark files can be improved by looking ahead a few steps. However it was noted that: "An optimal parsing scheme would also have to consider the possibility of matching a short first string and then a short second string in order to match a very long third string. We will however reject such possibilities as being too expensive to implement (for minimal expected gain in compression). Our non-greedy version of LZW will only look ahead by one parsed string when choosing its course of action." In fact the algorithm proposed in [Hor95] not only changes the parsing scheme but also constructs an entirely different dictionary from that of LZW on a given string – hence compression improvement over LZW is not guaranteed, and the notion of optimality is not clear. The worst case running time of this algorithm is $O(|T^{3/2}|)$, where $|T|$ is the input size.

An interesting fact for static dictionaries is that greedy parsing is optimal for those dictionaries with the suffix property [CK96]. It follows that if all the input is available off-line, it can be parsed optimally via a rigt-to-left greedy parsing provided that the dictionary is static and has the prefix property.

Recently [MS99] demonstrated that *for all dictionary construction schemes with the prefix property, greedy parsing with a single step lookahead is optimal on all input strings* – this scheme is called *flexible parsing* or $\mathcal{FP}$. A new data structure which implements the algorithm that uses LZW dictionary construction with $\mathcal{FP}$ in $O(|T|)$ time and space proportional to the number of phrases in the dictionary is introduced as well. The space and time complexity of this data structure is comparable to that of the original LZW implementation, hence optimal compression can be achieved without any overhead in complexity. Note that suffix trees can also be used for this application [RPE81]. However the $O(|T|))$ space complexity of the suffix tree is expected to be much larger than the space complexity of the new data structure which is proportional to the number of output phrases.

In this study, we report an experimental evaluation of $\mathcal{FP}$ in the context of LZW dictionary construction scheme:
(1) We demonstrate that optimality of $\mathcal{FP}$ in the context of the LZW dictionary construction, denoted as LZW-$\mathcal{FP}$, translates into considerable improvement over greedy parsing in practice. We also consider the algorithm of [Hor95], which uses flexible dictionary construction, denoted here as FPA. The LZW-$\mathcal{FP}$ and FPA al-

gorithms are compared with UNIX `compress` (LZW) and `gzip` (LZ77). On the tested data files, both LZW-$\mathcal{FP}$ and FPA perform better, up to 20% improved compression, than UNIX `compress`. They are both inferior to `gzip` on small to moderate text files, such as in the Calgary corpus, but are typically superior to gzip for files larger than 1MB, and for non-textual data files of all sizes. For pseudo-random strings and DNA sequences, the improvement is up to 35%.
(2) We introduce a new data structure based on Karp-Rabin fingerprints [KR87] to efficiently implement $\mathcal{FP}$. Currently our algorithms run about $3-5$ times slower than `compress` which is the fastest among all algorithms, both during compression and decompression. We are in the process of improving our implementations and hence leave reporting on explicit timing results to the full paper.
(3) We investigate whether better asymptotic properties of LZ78 based algorithms in comparison to LZ77 translate into improved compression. We demonstrate that on pseudorandom bit streams (with various distributions) the redundancy in the output of each of the four programs approach to the expected asymptotic behavior very fast – requiring less than 1KB for each of the different distributions; better asymptotic properties of LZW in comparison to LZ77 is very visible.[1] For files of size $1MB$, `compress` can improve over `gzip` up to 20% in compression achieved [2].

2 The Compression Algorithms

In this section we describe how each of the algorithms we consider work. We give the descriptions of the standard algorithms as well as new ones for the sake of completeness. Each of the algorithms fit in a general framework that we describe below.
Model. We denote a compression algorithm by $\mathcal{C}$, and its corresponding decompression algorithm by $\mathcal{C}^{\leftarrow}$. The input to $\mathcal{C}$ is a string T, of n characters, chosen from a constant size alphabet Σ; in our experiments Σ is either ascii or is $\{0, 1\}$. We denote by $T[i]$, the i^{th} character of T $(1 \leq i \leq n)$, and by $T[i : j]$ the substring which begins at $T[i]$ and ends at $T[j]$; notice that $T = T[1 : n]$.

The compression algorithm $\mathcal{C}$ compresses the input by reading the input charac-

[1] The average number of bits output by LZ78 or LZW, for the first n characters of an input string created by an i.i.d. source is only $O(1/\log n)$ more than its entropy [JS95, LS95]. A similar result for more general, unifilar, sources has been obtained by Savari [Sav97]. For the LZ77 algorithm, this redundancy is as much as $O(\log\log n/\log n)$ [Wyn95].

[2] All the software, documentation, and detailed experimental results reported in this paper are available on the WWW [Sou].

ters from left to right (i.e. from $T[1]$ to $T[n]$) and by partitioning it into substrings which are called blocks. Each block is replaced by a corresponding label that we call a codeword. We denote the j^{th} block by $T[b_j : b_{j+1} - 1]$, or shortly T_j, where $b_1 = 1$. The output of $\mathcal{C}$, hence, consists of codewords $C[1], C[2], \ldots, C[k]$ for some k, which are the codewords of blocks $T_1, T_2, \ldots, T_k$ respectively.

The algorithm $\mathcal{C}$ maintains a dynamic set of substrings called the dictionary, $\mathcal{D}$. Initially, $\mathcal{D}$ consists of all one-character substrings possible. The codewords of such substrings are their characters themselves. As the input T is read, $\mathcal{C}$ adds some of its substrings to $\mathcal{D}$ and assigns them unique codewords. We call such substrings of T phrases of $\mathcal{D}$. Each block T_j is identical to a phrase in $\mathcal{D}$: hence $\mathcal{C}$ achieves compression by replacing substrings of T with pointers to their earlier occurrences in T.

The decompression algorithm $\mathcal{C}^{\leftarrow}$ that corresponds to $\mathcal{C}$, takes $C[1 : k]$ as input and computes $T[1 : n]$ by replacing each $C[j]$ by its corresponding block T_j. Because the codeword $C[j]$ is a function of $T[1 : b_j - 1]$ only, the decompression can be correctly performed in an inductive fashion.

Below, we provide detailed descriptions of the compression algorithms considered, both the new and the old for the sake of completeness .

LZ-77 Algorithm. The LZ-77 algorithm reads the input characters from left to right while inserting all its substrings in $\mathcal{D}$. In other words, at the instance it reads $T[i]$, all possible substrings of the form $T[j : \ell]$, $j \leq \ell < i$ are in $\mathcal{D}$, together with all substrings of size one. The codeword of the substring $T[j : \ell]$, is the 2-tuple, $(i - j, \ell - j + 1)$, where the first entry denotes the relative location of $T[j : \ell]$, and the second entry denotes its size. LZ77 uses greedy parsing: the m^{th} block $T_m = T[b_m : b_{m+1} - 1]$ is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1} - 1]$.

LZW Algorithm. The LZW algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : b_{m+1}]$. Hence the phrases of LZW are the substrings obtained by concatenating the blocks of T with the next character following them, together with all possible substrings of size one. The codeword of the phrase $T[b_m : b_{m+1}]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. Thus, the codewords of substrings do not change in LZW algorithm. LZW uses greedy parsing as well: the m^{th} block T_m is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1} - 1]$. Hence, no two phrases can be identical in the LZW algorithm.

LZW-$\mathcal{FP}$ Algorithm. The LZW-$\mathcal{FP}$ algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b'_m : b'_{m+1}]$, where b'_m denotes the beginning location of block m if the compression algorithm used were

LZW. Hence for dictionary construction purposes LZW-$\mathcal{FP}$ emulates LZW: for any input string LZW and LZW-$\mathcal{FP}$ build identical dictionaries. The output generated by these two algorithms however are quite different. The codeword of the phrase $T[b'_m : b'_{m+1}]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. LZW-$\mathcal{FP}$ uses flexible parsing: intuitively, the m^{th} block T_m is recursively defined as the substring which results in the longest advancement in the next iteration. More precisely, let the function f be defined on the characters of T such that $f(i) = \ell$ where $T[i : \ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\ell]$. Then, given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\alpha - 1]$.

FPA Algorithm. The FPA algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : f(b_m)]$, where the function f is as described in LZW-$\mathcal{FP}$ algorithm. Hence for almost all input strings, FPA constructs an entirely different dictionary with that of LZW-$\mathcal{FP}$. The codeword of the phrase $T[b_m : f(b_m)]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. FPA again uses flexible parsing: given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$.

3 Data Structures and Implementations

In this section we describe both the trie-reverse-trie data structure, and the new fingerprints based data structure for efficient on-line implementations of the LZW-$\mathcal{FP}$, and FPA methods. The trie-reverse-trie pair is a deterministic data structure, and hence guarantees a worst case linear running time for both algorithms as described in [MS99]). The new data structure based on *fingerprints* [KR87], is randomized, and guarantees an expected linear running time for the algorithm.

The two main operations to be supported by these data structures are (1) insert a phrase to $\mathcal{D}$ (2) search for a phrase, i.e., given a substring S, check whether it is in $\mathcal{D}$. The standard data structure used in many compression algorithms including LZW, the compressed trie $\mathcal{T}$ supports both operations in time proportional to $|S|$. A compressed trie is a rooted tree with the following properties: (1) each node with the exception of the root represents a dictionary phrase; (2) each edge is labeled with a substring of characters; (3) the first characters of two sibling edges can not be identical; (4) the concatenation of the substrings of the edges from the root to a given node is the dictionary phrase represented by that node; (5) each node is labeled by the codeword corresponding to its phrase. Dictionaries with prefix properties, such as the ones used in LZW and LZ78 algorithms, build a regular trie rather than a compressed one. The only difference is that in a regular trie the substrings of all edges are one character long.

In our data structures, inserting a phrase S to $\mathcal{D}$ takes $O(|S|)$ time as in the case of a trie. Similarly, searching S takes $O(|S|)$ time if no information about substring S is provided. However, once it is known that S is in $\mathcal{D}$, searching strings obtained by concatenating or deleting characters to/from both ends of S takes only $O(1)$ time. More precisely, our data structures support two operations *extend* and *contract* in $O(1)$ time. Given a phrase S in $\mathcal{D}$, the operation extend(S, a) for a given character a, finds out whether the concatenation of S and a is a phrase in $\mathcal{D}$. Similarly, the operation contract(S), finds out whether the suffix $S[2 : |S|]$ is in $\mathcal{D}$. Notice that such operations can be performed in a suffix tree, if the phrases in $\mathcal{D}$ are all the suffixes of a given string as in the case of the LZ77 algorithm [RPE81]. For arbitrary dictionaries (such as the ones built by LZW) our data structures are unique in supporting contract and extend operations in $O(1)$ time, and insertion operation in time linear with the size of the phrase, while using $O(|\mathcal{D}|)$ space, where $|\mathcal{D}|$ is the number of phrases in $\mathcal{D}$.

Trie-reverse-trie-pair data structure. Our first data structure builds the trie, $\mathcal{T}$, of phrases as described above. In addition to $\mathcal{T}$, it also constructs $\mathcal{T}^r$, the compressed trie of the *reverses* of all phrases inserted in the $\mathcal{T}$. Given a string $S = s_1, s_2, \ldots, s_n$, its reverse S^r is the string $s_n, s_{n-1}, \ldots, s_2, s_1$. Therefore for each node v in $\mathcal{T}$, there is a corresponding node v^r in $\mathcal{T}^r$ which represents the reverse of the phrase represented by v. As in the case of the $\mathcal{T}$ alone, the insertion of a phrase S to this data structure takes $O(|S|)$ time. Given a dictionary phrase S, and the node n which represents S in $\mathcal{T}$, one can find out whether the substring obtained by concatenating S with any character a in is $\mathcal{D}$, by checking out if there is an edge from n with corresponding character a; hence extend operation takes $O(1)$ time. Similarly the contract operation takes $O(1)$ time by going from n to n', the node representing reverse of S in $\mathcal{T}^r$, and checking if the parent of n' represents $S[2 : |S|]^r$.

Fingerprints based data structure. Our second data structure is based on building a hash table H of size p, a suitably large prime number. Given a phrase $S = S[1 : |S|]$, its location in H is computed by the function h, where $h(S) = (s[1]|\Sigma|^{|S|} + s[2]|\Sigma|^{|S|-1} + \ldots + s[|S|]) \, mod \, p$, where $s[i]$ denotes the lexicographic order of $S[i]$ in Σ [KR87]. Clearly, once the values of $|\Sigma|^k \, mod \, p$ are calculated for all k up to the maximum phrase size, computation of $h(S)$, takes $O(|S|)$ time. By taking p sufficiently large, one can decrease the probability of a collision on a hash value to some arbitrarily small $1/\epsilon$ value; thus the average running time of an insertion would be

$O(|S|)$ as well. Given the hash value $h(S)$ of a string, the hash value of its extension by any character a can be calculated by $h(Sa) = (h(S)|\Sigma| + lex(a))\ mod\ p$, where $lex(a)$ is the lexicographic order of a in Σ. Similarly, the hash value of its suffix $S[2 : |S|]$ can be calculated by $h(S[2 : |S|]) = (h(S) - s[1]|\Sigma|^{|S|})\ mod\ p$. Both operations take $O(1)$ time.

In order to verify if the hash table entry $h(S)$ includes S in $O(1)$ time we (1) give unique labels to each of the phrases in $\mathcal{D}$, and (2) in each phrase S in H, store the label of the suffix $S[2 : |S|]$ and the label of the prefix $S[1 : |S| - 1]$. The label of newly inserted phrase can be $|\mathcal{D}|$, the size of the dictionary. This enables both extend and contract operations to be performed in $O(1)$ time on the average: suppose the hash value of a given string S is h, and the label of S is ℓ. To extend S with character a, we first compute the hash value h' of the string Sa. Among the phrases whose hash value is h', the one whose prefix label matches the label of S gives the result of the extend operation. To contract S, we first compute the hash value h'' of the string $S[2 : |S|]$. Among the phrases whose hash value is h'', the one whose label matches the suffix label of S gives the result of the contract operation. Therefore, both extend and contract operations take expected $O(1)$ time.

Inserting a phrase in this data structure can be performed as follows. An insert operation is done only after an extend operation on some phrase S (which is in $\mathcal{D}$) with some character a. Hence, when inserting the phrase Sa in $\mathcal{D}$ its prefix label is already known: the label of S. Once it is decided that Sa is going to be inserted, we can spend $O(|S| + 1)$ time to compute the suffix label of Sa. In case the suffix $S[2 : |S|]a$ is not a phrase in $\mathcal{D}$, we temporarily insert an entry for $S[2 : |S|]a$ in the hash table. This entry is then filled up when $S[2 : |S|]$ is actually inserted in $\mathcal{D}$. Clearly, the insertion operation for a phrase R takes expected $O(|R|)$ time.

A linear time implementation of LZW-$\mathcal{FP}$. For any input T LZW-$\mathcal{FP}$ inserts to $\mathcal{D}$ the same phrases with LZW. The running time for insertion in both LZW and LZW-$\mathcal{FP}$ (via the data structures described above) are the same; hence the total time needed to insert all phrases in LZW-$\mathcal{FP}$ should be identical to that of LZW, which is linear with the input size. Parsing with $\mathcal{FP}$ consists of a series of extend and contract operations. We remind that: (1) the function f on characters of T is described as $f(i) = \ell$ where $T[i : \ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$. (2) given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$. In order to compute b_{m+1}, we inductively assume that $f(b_m)$ is already computed. Clearly $S = T[b_m : f(b_m)]$ is in $\mathcal{D}$ and $S' = T[b_m : f(b_m)]$ is not in $\mathcal{D}$. We then contract S by i characters, until $S' = T[b_m + i : f(b_m) + 1]$ is in $\mathcal{D}$. Then we proceed with extensions to compute $f(b_m + i)$. After subsequent contract and extends we stop once $i > f(b_m)$. The last value of i at which we started our final round of contracts is the value b_{m+1}. Notice that each character in T participates to exactly one extend and one contract operation, each of which takes $O(1)$ time via the data structures described above. Hence the total running time for the algorithm is $O(n)$.

4 Experiments

In this section we describe in detail the data sets we used, and discuss our test results verifying how well our theoretical expectations were supported.

The test programs. We used `gzip`, `compress`, LZW-$\mathcal{FP}$ and FPA programs for our experiments. In our LZW-$\mathcal{FP}$ implementation we limited the dictionary size to 2^{16} phrases, and reset it when it was full as in the case of `compress`; we also experimented with the extended version of LZW-$\mathcal{FP}$ which allows 2^{24} phrases. Similarly we experimented with two versions of FPA: one with 2^{16} and the other with 2^{24} phrases maximum.

The data sets. Our data sets come from three sources: (1) Data obtained via UNIX `drand48()` pseudorandom number generator - designed to measure the asymptotic redundancy in algorithms. (2) DNA and protein sequences provided by Center for BioInformatics, University of Pennsylvania and CT and MR scans provided by the St. Thomas Hospital, UK [Sou]. (3) Text files from two data compression benchmark suites: the new Canterbury corpus and the commonly used Calgary corpus [Sou].

Specifically, the first data set includes three binary files generated by the UNIX `drand48()` function. The data distribution is i.i.d. with bit probabilities (1) $0.7 - 0.3$, (2) $0.9 - 0.1$, and (3) $0.97 - 0.03$. The second data set includes two sets of human DNA sequences from chromosome 23 (*dna1, dna2*), one MR (magnetic resonance) image of human (female) breast (*mr.pgm*), and one CT (computerized tomography) scan of a fractured human hip *ct.pgm* in uncompressed `pgm` format in ASCII [Sou]. The third set includes the complete Calgary corpus; the corresponding table is ommited here for lack of space, and can be found at [Sou]. It also includes all files of size $> 1MB$ from the new Canterbury corpus: a DNA sequence from E-coli bacteria, *E.coli,* the complete bible *bib.txt* , and *world192.txt.*

Test results. In summary, we observed that LZW-$\mathcal{FP}$ and FPA implementations with maximum dictionary size 2^{24} performs the best on all types of files with size $> 1MB$ and shorter files with non-textual content. For shorter files consisting text, `gzip` performs the best as expected.

Our tests on the human DNA sequences with LZW-$\mathcal{FP}$ and FPA show similar improvements over *compress* and *gzip* - with a dictionary of maximum size 2^{16}, the improvement is about 1.5% and 5.7% respectively. Some more impressive results were obtained by increasing the dictionary size to 2^{24}, which further improved the compression ratio to 9%. The performance of LZW-$\mathcal{FP}$ and FPA on *mr* and *ct* scans differ quite a bit: LZW-$\mathcal{FP}$ was about $4\% - 6\%$ better than `compress` and was comparable to `gzip`; FPA's improvement was about 15% and 7% respectively. As the image files were rather short, we didn't observe any improvement by using a larger dictionary. One interesting observation is that the percentage improvement achieved by both FPA and LZW-$\mathcal{FP}$ increased consistently with increasing data size. This suggests that we can expect them to perform better in compressing massive archives as needed in many biomedical applications such as the human genome project.

Our results on text strings varied depending on the type and size of the file compressed. For short files with long repetitions, `gzip` is still the champion. However, for all text files of size $> 1MB$, the large dictionary implementation of FPA scheme outperforms `gzip` by $4.7\% - 8.5\%$, similar to the tests for DNA sequences. The following tables demonstrate the relative performances of the test programs on the data sets: Column 1 shows original file size (with some prefixes), column 2 and column 3 show the compressed file size by *gzip* and *compress* respectively, and the remaining columns show the improvement (%) made by *LZW-FP*, *FPA*, *FP-24*, and *FPA-24* over *gzip* and *compress*.

File	*Size*	*gzip*	*comprs*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
E.coli	4530	1341	1255	6.91	0.56	6.43	0.05	8.84	2.63	8.48	2.24
bible.txt	3953	1191	1401	-12.87	4.11	-7.79	8.42	0.13	15.15	4.68	19.01
world192.txt	2415	724	987	-31.70	3.32	-20.36	11.64	-2.38	24.84	6.54	31.39

Table 1: Compression evaluation using files in the Canterbury corpus (Large Set)

References

[CK96] M. Cohn and R. Khazan. Parsing with suffix and prefix dictionaries. In *IEEE Data Compression Conference*, 1996.

[GSS85] M. E. Gonzales-Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, 32(2):344–373, April 1985.

[Hor95] R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *IEEE Data Compression Conference*, 1995.

File	*Size*	*gzip*	*comprs*	*LZW-$\mathcal{FP}$*		*FPA*		*FP-24*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
P(0)=0.7	1	.2	.2	4.88	6.25	4.88	6.25	4.88	6.25	4.88	6.25
	100	15.7	13.4	15.60	1.62	17.89	4.29	15.61	1.64	17.89	4.29
	2048	**320**	**263**	**18.53**	**1.07**	**20.46**	**3.41**	**19.44**	**2.17**	**21.20**	**4.31**
P(0)=0.9	1	.1	.1	3.10	6.72	9.30	12.69	3.10	6.72	9.30	12.69
	100	9.7	7.75	22.73	2.89	25.86	6.83	22.74	2.90	25.86	6.83
	2048	**200**	**147**	**28.10**	**2.07**	**30.95**	**5.95**	**28.43**	**2.50**	**31.20**	**6.28**
P(0)=0.97	1	.09	.1	6.45	12.12	6.45	12.12	6.45	12.12	6.45	12.12
	100	4.6	3.8	22.39	4.42	28.03	11.37	22.41	4.45	28.03	11.37
	2048	**91.4**	**64.8**	**31.30**	**3.10**	**35.79**	**9.44**	**31.30**	**3.11**	**35.79**	**9.44**

Table 2: Compression evaluation using independent identically distributed random files containing only zeros and ones with different probability distributions

File	*Size*	*gzip*	*comprs*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
dna1	3096	977	938	5.59	1.54	5.75	1.70	8.73	4.82	8.91	5.00
dna2	2877	846	813	4.64	0.75	4.33	0.43	6.09	2.26	5.89	2.05
mr.pgm	260	26	29	-7.23	3.60	6.38	15.84	-7.22	3.61	6.38	15.84
ct.pgm	1039	110	110	4.10	3.61	14.56	14.12	4.10	3.61	14.56	14.12

Table 3: Compression evaluation using experimental biological and medical data

[JS95] P. Jacquet and W. Szpankowski. Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees. *Theoretical Computer Science,* (144):161–197, 1995.

[KR87] R. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[LS95] G. Louchard and W. Szpankowski. Average profile and limiting distribution for a phrase size in the Lempel-Ziv parsing algorithm. *IEEE Transactions on Information Theory*, 41(2):478–488, March 1995.

[MS99] Y. Matias and S. C. Sahinalp. On optimality of parsing in dynamic dictionary based data compression. ACM-SIAM Symposium on Discrete Algorithms, 1999.

[RPE81] M. Rodeh, V. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM,* 28(1):16–24, January 1981.

[Sav97] S. Savari. Redundancy of the Lempel-Ziv incremental parsing rule. In *IEEE Data Compression Conference*, 1997.

[Sou] http://www.dcs.warwick.ac.uk/people/research/Nasir.Rajpoot/work/fp/index.html.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer,* pages 8–19, January 1984.

[Wyn95] A. J. Wyner. *String Matching Theorems and Applications to Data Compression and Statistics*. Ph.D. dissertation, Stanford University, Stanford, CA, 1995.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.